View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2011 by Shivaram Venkataraman. All rights reserved.

STORAGE SYSTEM DESIGN FOR NON-VOLATILE BYTE-ADDRESSABLE MEMORY USING CONSISTENT AND DURABLE DATA STRUCTURES

BY

SHIVARAM VENKATARAMAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

Abstract

The predicted shift to low cost, non-volatile, byte-addressable memory (e.g., Phase Change Memory and Memristor), the growth of "big data", and the subsequent emergence of frameworks such as memcached and NoSQL systems require us to rethink the design of storage systems. Today, the predominant method available to application programmers for obtaining persistence of data is through the file interface which provides access to a filesystem, which in turn uses the block layer to provide access to hard disk drives or solid state disks. With the advent of non-volatile byte-addressable memory (NVBM), buffered I/O systems designed primarily for hard disks are no longer the most efficient design for storage. We show that the overheads imposed by system calls, the file interface, and the block device layer can make NVBM access up to 266 times slower compared to directly performing loads and stores.

With performance characteristics closer to DRAM, NVBM demands software designs that resemble common heap data structures rather than current systems that serialize data via the file interface. To derive the maximum performance, we propose the use of single-level data stores where no distinction is made between a volatile and a persistent copy of data. For these systems, we present Consistent and Durable Data Structures (CDDSs) that, on current hardware, allows programmers to safely exploit the low-latency and non-volatile aspects of new memory technologies. CDDSs use versioning to allow atomic updates without requiring logging. The same versioning scheme also enables rollback for failure recovery. When compared to a memory-backed Berkeley DB B-Tree, our prototype-based results show that a CDDS B-Tree can increase *put* and *get* throughput by 74% and 138%. Tembo, a CDDS B-Tree enabled distributed Key-Value system, has up to 250%–286% higher throughput than Cassandra, a two-level data store.

To my

Mother and Father

Acknowledgments

I would like to thank my advisor Roy H. Campbell for his invaluable guidance over the course of my Masters degree and for giving me the opportunity to be a part of the Systems Research Group. This has helped me learn a lot about academic research. I am deeply indebted to Niraj Tolia for encouraging me to work on the design of storage systems for non-volatile memory and much of the work described in this thesis was born out of our discussions. I would also like to thank Matthew Caesar for his encouragement and Parthasarathy Ranganathan, Nathan L. Binkert for their valuable inputs towards this work.

I would like to thank Ellick Chan, Abhishek Verma and Reza Farivar for working with me on multiple research projects over the last two years. I am also thankful to Alejandro Gutierrez, Mirko Montanari, Cristina Abad, Kevin Larson and Will Dietz for the many technical discussions, and all the members of Systems Research Group for the vibrant research environment.

My research has been funded by a fellowship from the Siebel Scholars Foundation and internships at HP Labs, Palo Alto. I would also like to thank Cinda Heeren for the invaluable skills I learnt and the wonderful experience I had as a Teaching Assistant. Lynette Llubben, the staff at the Computer Science department and the staff at HP Labs have helped me with many administrative tasks and I thank them for their help. My time at Urbana-Champaign has been made wonderful by support from many friends and family. Finally I would like to thank my parents for their constant support and for encouraging me to pursue my dreams.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Thesis Statement: Data structures can be re-designed to provide consistent and durable updates on non-volatile byte-addressable memory using multi-versioning techniques.

Recent architecture trends and our conversations with memory vendors show that DRAM density scaling is facing significant challenges and will hit a scalability wall beyond 40nm [34, 42, 44]. Additionally, power constraints will also limit the amount of DRAM installed in future systems [7, 27]. To support next generation systems, including large memory-backed data stores such as memcached [26] and RAMCloud [49], technologies such as Phase Change Memory [52] and Memristor [62] hold promise as DRAM replacements. Because these memory technologies are non-volatile and their cost/power-efficiency is favorable, trends indicate that NVBM can also replace disks as the primary storage medium. The latency for accessing NVBM will be orders of magnitude faster than disk or flash and we believe that NVBM will provide a single storage layer which has disk-like capacity with DRAM-like performance [27].

Current operating systems use file systems to provide support for storage on persistent media like HDDs or SSDs. File systems are accessed through the *file interface* (open/read/write system calls) and provide support for consistent storage of data across failures. File systems then send data to the buffer cache, which buffers writes and facilitates asynchronous flushing of data to the disk through the block layer. The block layer schedules I/O operations and is optimized for disks. There are examples of systems that bypass the file system interface, such as databases, but they, like file systems, still use a buffer cache tuned for disks and send data to the block layer.

The low latency access times of NVBM means that such legacy interfaces will not be suitable for applications that require high performance. First, the overhead of PCI accesses or system calls dominate NVBM's sub-microsecond access latencies. More importantly, the file interface imposes a programming model where data needs to be serialized and updated at periodic intervals. This imposes a two-level logical separation of data, differentiating between in-memory and on-disk copies. Traditional data stores have to both update the in-memory data and, for durability, sync the data to disk with the help of a write-ahead log. Not only does this data movement use extra power [7] and reduce performance for low-latency NVBM, the logical separation also reduces the usable capacity of an NVBM system. We believe that the block and file system layers are not suitable for accessing NVBM because of the following reasons:

- Performance: The overhead for updating a byte on persistent storage is significant compared to NVBM access latency. It should also be noted that this is not a new problem and has been observed before by the database community [60] where systems like System R and INGRES maintained a separate buffer pool in userspace to reduce the overhead.
- Programming model: The file interface establishes that persistent data should be accessed using read() and write() calls. Applications serialize data that needs to be stored and have to deserialize the data into in-memory buffers before using it. This imposes CPU overhead for serializing / deserializing and also imposes a programming model where multiple copies need to be maintained.
- Durability: Projected designs for integrating NVBM indicate that it can be made available through DIMM slots. This means that the granularity of updates to the device can be the size of a cache line, which is much smaller than the size of a hard-disk sector. The block device layer which flushes data on a page-level granularity is not efficient for making small updates to NVBM.

We measure the overhead imposed by each of the intermediate layers in Chapter 3 and find that the memory bandwidth for serial writes of large blocks is 6x higher when updates are made directly to memory rather than through the file system and block device layer. This difference balloons to 266x when writes are cache-line sized. Instead, we propose a single-level NVBM hierarchy where applications can directly perform loads and stores on non-volatile memory and where no distinction is made between a volatile and persistent copy of data. In particular, we propose the use of Consistent and Durable Data Structures (CDDSs) to store data, a design that allows for the creation of log-less systems on non-volatile memory without processor modifications. Described in Chapter 4, these data structures allow mutations to be safely performed directly (using loads and stores) on the single copy of the data and metadata. We have architected CDDSs to use versioning. Independent of the update size, versioning allows the CDDS to atomically move from one consistent state to the next, without the extra writes required by logging or shadow paging. Failure recovery simply restores the data structure to the most recent consistent version. Further, while complex processor changes to support NVBM have been proposed [19], we show how primitives to provide durability and consistency can be created using existing processors.

We have implemented a CDDS B-Tree because of its non-trivial implementation complexity and widespread use in storage systems. Our evaluation, presented in Chapter 5, shows that a CDDS B-Tree can increase put and get throughput by 74% and 138% when compared to a memory-backed Berkeley DB B-Tree. Tembo[1], our Key-Value (KV) store described in Chapter 4.6, was created by integrating this CDDS B-Tree into a widely-used open-source KV system. Using the Yahoo Cloud Serving Benchmark [20], we observed that Tembo increases throughput by up to 250%–286% when compared to memory-backed Cassandra, a two-level data store. Further, in order to compare

[1] Swahili for elephant, an animal anecdotally known for its memory.

different approaches to providing durable updates, we use Cafegrind [13], a tool which monitors a running program and tracks the usage of dynamically allocated data structures. We analyze how versioning could help build integrated wear-leveling solutions for NVBM by preventing hot spots. We also compare write-ahead logging and versioning in terms of the memory bandwidth used over time.

Chapter 2

Background

2.1 Hardware Non-Volatile Memory

Significant changes are expected in the memory industry. Non-volatile flash memories have seen widespread adoption in consumer electronics and are starting to gain adoption in the enterprise market [28]. Recently, new NVBM memory technologies (e.g., PCM, Memristor, and STTRAM) have been demonstrated that significantly improve latency and energy efficiency compared to flash.

As an illustration, we discuss Phase Change Memory (PCM) [52], a promising NVBM technology. PCM is a non-volatile memory built out of Chalcogenide-based materials (e.g., alloys of germanium, antimony, or tellurium). Unlike DRAM and flash that record data through charge storage, PCM uses distinct phase change material states (corresponding to resistances) to store values. Specifically, when heated to a high temperature for an extended period of time, the materials crystallize and reduce their resistance. To reset the resistance, a current large enough to melt the phase change material is applied for a short period and then abruptly cut-off to quench the material into the amorphous phase. The two resistance states correspond to a '0' and '1', but, by varying the pulse width of the reset current, one can partially crystallize the phase change material and modify the resistance to an intermediate value between the '0' and '1' resistances. This range of resistances enables multiple bits per cell, and the projected availability of these MLC designs is 2012 [33].

Table 2.1 summarizes key attributes of potential storage alternatives in the next decade, with projected data from recent publications, technology trends, and direct industry communication. These trends suggest that future non-volatile memories such as PCM or Memristors can be viable DRAM replacements, achieving competitive speeds with much lower power consumption, and with non-volatility properties similar to disk but without the power overhead. Additionally, a number of recent studies have identified a slowing of DRAM growth [33, 34, 39, 42, 44, 50, 70] due to scaling challenges for charge-based memories. In conjunction with DRAM's power inefficiencies [7, 27], these trends can potentially accelerate the adoption of NVBM memories.

NVBM technologies have traditionally been limited by density and endurance, but recent trends suggest that these limitations can be addressed. Increased density can be achieved within a single-die through multi-level designs, and,

Technology	Density	Read/Write Latency		Read/Write Energy		Endurance
	μm^2/bit	ns		pJ/bit		writes/bit
HDD	0.00006	3,000,000	3,000,000	2,500	2,500	∞
Flash SSD (SLC)	0.00210	25,000	200,000	250	250	10^5
DRAM (DIMM)	0.00380	55	55	24	24	10^{18}
PCM	0.00580	48	150	2	20	10^8
Memristor	0.00580	100	100	2	2	10^8

Table 2.1: Non-Volatile Memory Characteristics: 2015 Projections

potentially with multiple-layers per die. At a single chip level, 3D die stacking using through-silicon vias (TSVs) for inter-die communication can further increase density. PCM and Memristor also offer higher endurance than flash (10^8 writes/cell compared to 10^5 writes/cell for flash). Optimizations at the technology, circuit, and systems levels have been shown to further address endurance issues, and more improvements are likely as the technologies mature and gain widespread adoption.

These trends, combined with the attributes summarized in Table 2.1, suggest that technologies like PCM and Memristors can be used to provide a single "unified data-store" layer - an assumption underpinning the system architecture in our paper. Specifically, we assume a storage system layer that provides disk-like functionality but with memory-like performance characteristics and improved energy efficiency. This layer is persistent and byte-addressable. Additionally, to best take advantage of the low-latency features of these emerging technologies, non-volatile memory is assumed to be accessed off the memory bus. Like other systems [17, 19], we also assume that the hardware can perform atomic 8 byte writes.

While our assumed architecture is future-looking, it must be pointed out that many of these assumptions are being validated individually. For example, PCM samples are already available (e.g., from Numonyx) and an HP/Hynix collaboration [30] has been announced to bring Memristor to market. In addition, aggressive capacity roadmaps with multi-level cells and stacking have been discussed by major memory vendors. Finally, previously announced products have also allowed non-volatile memory, albeit flash, to be accessed through the memory bus [59].

2.2 File Systems and Object Stores

Traditional disk-based file systems are also faced with the problem of performing atomic updates to data structures. File systems like WAFL [31] and ZFS [63] use shadowing to perform atomic updates. Failure recovery in these systems is implemented by restoring the file system to a consistent snapshot that is taken periodically. These snapshots are created by shadowing, where every change to a block creates a new copy of the block. Recently, Rodeh [54] presented a B-Tree construction that can provide efficient support for shadowing, and this technique has been used in the design of BTRFS [48]. Failure recovery in a CDDS uses a similar notion of restoring the data structure to the most recent

consistent version. However the versioning scheme used in a CDDS results in fewer data-copies when compared to shadowing.

Arguments for limiting the role of operating systems are not new. Microkernels and Exokernels [24] proposed operating system primitives which allow applications to interact more directly with hardware. This is similar to our design where we bypass the file system and block device layers for I/O operations. Database systems [60] have also used user space buffer pools and direct unbuffered writes to devices to avoid some of the overheads outlined in this work. We believe that removing the operating system from the critical path of I/O operations will help build better performing higher-level data storage systems. Systems like Lightweight Recoverable Virtual Memory [56] and QuickStore [68] have a similar goal and provide support for persistent objects on top virtual memory pages while using disks as a backing store. Earlier efforts at designing persistent operating systems [21] have also identified the complexity of serializing data for persistent storage on file systems.

2.3 Non-Volatile Memory-based Systems

The use of non-volatile memory to improve performance is not new. eNVy [69] designed a non-volatile main memory storage system using flash. eNVy, however, accessed memory on a page-granularity basis and could not distinguish between temporary and permanent data. The Rio File Cache [15, 41] used battery-backed DRAM to emulate NVBM but it did not account for persistent data residing in volatile CPU caches. Recently there have been many efforts [29] to optimize data structures for flash memory based systems. FD-Tree [40] and BufferHash [3] are examples of write-optimized data structures designed to overcome high-latency of random writes, while FAWN [4] presents an energy efficient system design for clusters using flash memory. However, design choices that have been influenced by flash limitations (e.g., block addressing and high-latency random writes) render these systems suboptimal for NVBM.

Qureshi et al. [50] have also investigated combining PCM and DRAM into a hybrid main-memory system but do not use the non-volatile features of PCM. While our work assumes that NVBM wear-leveling happens at a lower layer [70], it is worth noting that versioning can help wear-leveling as frequently written locations are aged out and replaced by new versions. Most closely related is the work on NVTM [17] and BPFS [19]. NVTM, a more general system than CDDS, adds STM-based [57] durability to non-volatile memory. However, it requires adoption of an STM-based programming model. Further, because NVTM only uses a metadata log, it cannot guarantee failure atomicity. BPFS, a PCM-based file system, also proposes a single-level store. However, unlike CDDS's exclusive use of existing processor primitives, BPFS depends on extensive hardware modifications to provide correctness and durability. Further, unlike the data structure interface proposed in this work, BPFS implements a file system interface. While this is transparent to legacy applications, the system-call overheads reduce NVBM's low-latency benefits.

2.4 Data Store Trends

The growth of "big data" [1] and the corresponding need for scalable analytics has driven the creation of a number of different data stores today. Best exemplified by NoSQL systems [12], the throughput and latency requirements of large web services, social networks, and social media-based applications have been driving the design of next-generation data stores. In terms of storage, high-performance systems have started shifting from magnetic disks to flash over the last decade. Even more recently, this shift has accelerated to the use of large memory-backed data stores. Examples of the latter include memcached [26] clusters over 200 TB in size [37], memory-backed systems such as RAMCloud [49], in-memory databases [61, 66], and NoSQL systems such as Redis [53]. As DRAM is volatile, these systems provide data durability using backend databases (e.g., memcached/MySQL), on-disk logs (e.g., RAMCloud), or, for systems with relaxed durability semantics, via periodic checkpoints. We expect that these systems will easily transition from being DRAM-based with separate persistent storage to being NVBM-based.

Chapter 3

File System Architecture

Operating systems have traditionally provided support for persistent storage through file systems. Files are read (or updated) by making system calls which copy a region of memory from (or to) the device. Introduced in the Multics input/output system [25], the file interface is widely established and is also used to access other devices like network cards in UNIX based operating systems. The file system architecture in Linux is shown in Figure 3.1. Consider a write request for updating a particular region of a file. First, the request passes through the file system layer where changes to file system data structures like the inode map are made. Then the updated pages are buffered in the disk buffer cache. Pages marked as dirty in the disk buffer cache are then periodically flushed to the block device layer, which is used to schedule I/O operations for the hardware. The block device layer also supports buffered I/O where a block of data can be read, modified and then written back to the device. With the advent of non-volatile byte-addressable memory, we believe that the overhead imposed by these layers will be significant compared to the access latency for NVBM.

In order to measure the overhead at each layer, we perform an experiment where a region of memory is updated in blocks of different sizes. Since NVBM is not yet commercially available, we perform the experiments on DRAM, which has been shown to be a good indicator of NVBM performance [19]. All our experiments were performed on a server with two Intel Xeon Quad-Core 2.67 GHz (X5550) processors and 48 GB of 1333 MHz DDR3-RAM. Each processor has 128 KB L1, 256 KB L2, and 8 MB L3 caches. We used the Ubuntu 10.04 Linux distribution and the 2.6.32-24 64-bit kernel.

3.1 Direct Mapped Access

For measuring the overhead at each layer, we use a direct mapped experiment, where a region of memory is mapped into the application and updates are performed directly on it, as a baseline. In the baseline experiment there are no system calls invoked during the updates. Updates of different sizes are performed using `memcpy` and in order to ensure that there is no overhead due to page faults, we first write to the entire region before beginning the updates. The memory bandwidth obtained for different update sizes is shown in Figure 3.2 and Figure 3.3. The memory bandwidth

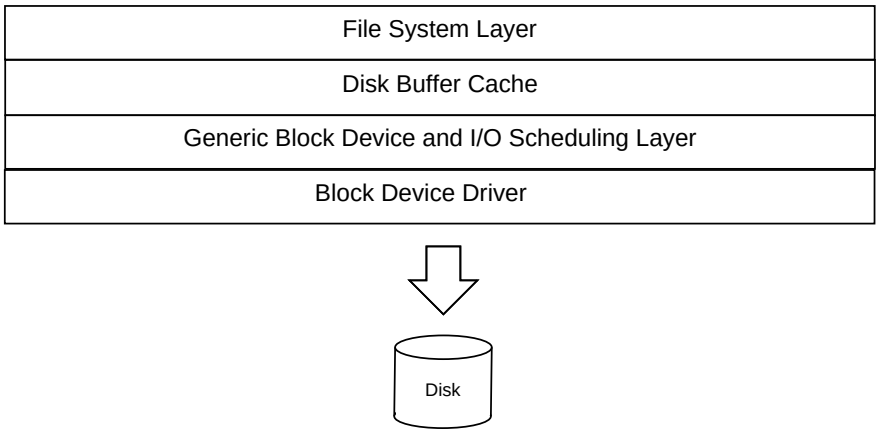


Figure 3.1: File system architecture

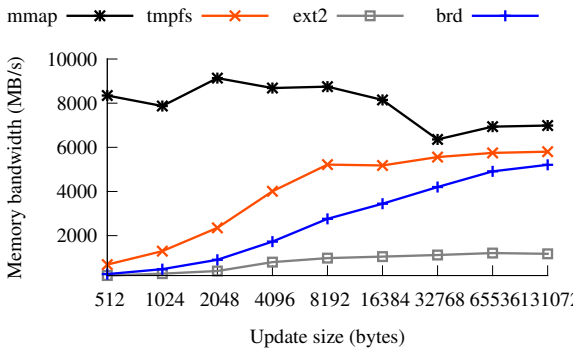


Figure 3.2: Block device and File system overhead for large updates. Note the linear scale on the y-axis

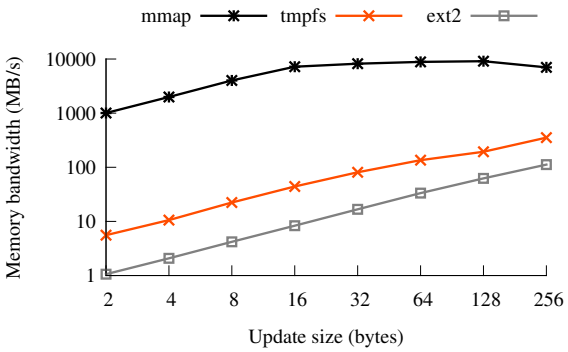


Figure 3.3: Block device and File system overhead for small updates. Note the log scale on the y-axis.

for direct mapped access increases as the size of the updates increase from 1GB/s for 2 byte updates to 9.1 GB/s for 128 byte updates. We believe this is related to the number of bytes that can be transferred using one SSE instruction.

3.2 Block Device Layer

To measure the overhead imposed by the block device layer, we measure the performance of using the Block RAM device (BRD) [35] to perform serial writes to a region of memory. The Block RAM device in Linux exposes main memory as a block device and we perform updates using write calls directly to the device, without mounting any file system on it. Also in order to ensure that the writes are not cached at the disk buffer cache, we open the device using the O_DIRECT flag. In this case, updates to the file are directly flushed to the block device. The results presented in Figure 3.2 show that as the block size is varied between 512 bytes and 128KB, the overhead of using the block device layer varies between 32x and 1.3x in terms of memory bandwidth. (We were unable to perform updates smaller than

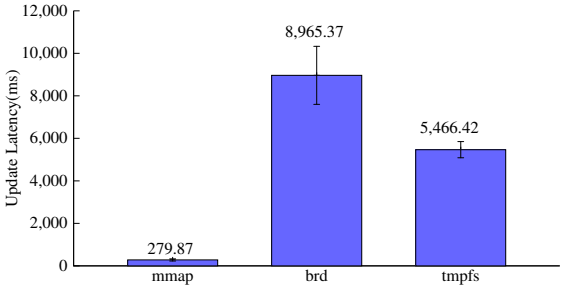


Figure 3.4: Latency for cache line sized update and flush

System	No. of function calls
tmpfs	64
Block RAM Device (BRD)	136
ext2 on BRD	192

Table 3.1: Number of kernel functions invoked in one `sys_write` system call

512 bytes using O_DIRECT as direct updates need to be multiples of the disk sector size. This is another indication that the block device layer is tuned for disks.)

3.3 File Systems

We consider two different configurations to demonstrate the overhead of using a file system and correspondingly the file interface. First, we mount ext2 on BRD and measure the time required to perform serial writes to a file. The results shown in Figure 3.3 show that mounting ext2 on BRD performs worse than writing directly to the device. This is due to the additional overhead of the file system layer which is combined with that of the block device interface.

We also considered `tmpfs` [16] an in-memory file system in Linux, where the data is stored in the page cache. As a result, there is no block device overhead in this case and we can measure the overhead of the file system layer. The results shown in Figure 3.3 show that for smaller 64B block sizes, direct mapped access is 65 times faster than using `tmpfs`. However as the block size is increased to 4K, we find that the performance converges to within 83% of direct mapped access.

3.4 Flush Overhead

Since NVBM will be available through DIMM slots, we can flush cache line sized updates to the device. However the block device and the file system layer can only flush updates which are equal to the page size (4KB in Linux). We performed an experiment to measure the overhead due to this limitation. In this experiment, we create a file of a fixed size and seek to a random location in the file. We then perform a cache line sized update and flush the update to NVBM using the `fsync` call. The Block RAM device and `tmpfs` layers were modified to invoke the `clflush` instruction for a sync operation. We compare the time taken against performing `clflush` from a userspace application. The results of this experiment are shown in Figure 3.4. The time taken to update and flush a cache line is around 280ns

from the userspace application. When compared to direct mapped access, the latency of using `tmpfs` and the Block RAM device is greater by 19.5x and 32x respectively.

3.5 System Call Overhead

In addition to measuring the memory bandwidth and latencies, we profiled the kernel functions invoked for every system call using LTTng [43], a Linux-kernel tracing tool. The number of function calls invoked for a single `sys_write` system call for `tmpfs`, BRD and ext2 on BRD is shown in Table 3.1. While the system call implementations have been optimized over the years, their performance will always be slower than performing no system calls at all. Hence, providing direct mapped access to NVBM removes the operating system from the critical path and overcomes any overheads for performing reads and writes. However, the OS will play in important role in providing isolation for directly mapped pages and we discuss issues related to safety in Chapter 7.

Chapter 4

Design and Implementation

As mentioned previously, we expect NVBM to be exposed across a memory bus and not via a legacy disk interface. In addition to the file system and block device overhead measured in Chapter 3, using the PCI interface (256 ns latency [32]) or even a kernel-based syscall API (89.2 and 76.4 ns for POSIX `read/write`) would add significant overhead to NVBM's access latencies (50–150 ns). As shown in Chapter 3, the overhead of using file systems or buffered updates would also be prohibitively large. Further, given the performance and energy cost of moving data, we believe that all data should reside in a single-level store where no distinction is made between volatile and persistent storage and all updates are performed in-place. We therefore propose that data access should use userspace libraries and APIs that map data into the process's address space.

However, the same properties that allow systems to take full advantage of NVBM's performance properties also introduce challenges. In particular, one of the biggest obstacles is that current processors do not provide primitives to order memory writes. Combined with the fact that the memory controller can reorder writes (at a cache line granularity), current mechanisms for updating data structures are likely to cause corruption in the face of power or software failures. For example, assume that a hash table insert requires the write of a new hash table object and is followed by a pointer write linking the new object to the hash table. A reordered write could propagate the pointer to main memory before the object and a failure at this stage would cause the pointer to link to an undefined memory region. Processor modifications for ordering can be complex [19], do not show up on vendor roadmaps, and will likely be preceded by NVBM availability.

To address these issues, our design and implementation focuses on three different layers. First, in Chapter 4.1, we describe how we implement ordering and flushing of data on existing processors. However, this low-level primitive is not sufficient for atomic updates larger than 8 bytes. In addition, we therefore also require versioning CDDSs, whose design principles are described in Chapter 4.2. After discussing our CDDS B-Tree implementation in Chapter 4.4 and some of the open opportunities and challenges with CDDS data structures in Chapter 4.5, Chapter 4.6 describes Tembo, the system resulting from the integration of our CDDS B-Tree into a distributed Key-Value system.

4.1 Flushing Data on Current Processors

As mentioned earlier, today's processors have no mechanism for preventing memory writes from reaching memory and doing so for arbitrarily large updates would be infeasible. Similarly, there is no guarantee that writes will not be reordered by either the processor or by the memory controller. While processors support a `mfence` instruction, it only provides write visibility and does not guarantee that all memory writes are propagated to memory (NVBM in this case) or that the ordering of writes is maintained. While cache contents can be flushed using the `wbinvd` instruction, it is a high-overhead operation (multiple ms per invocation) and flushes the instruction cache and other unrelated cached data. While it is possible to mark specific memory regions as write-through, this impacts write throughput as all stores have to wait for the data to reach main memory.

To address this problem, we use a combination of tracking recently written data and use of the `mfence` and `clflush` instructions. `clflush` is an instruction that invalidates the cache line containing a given memory address from all levels of the cache hierarchy, across multiple processors. If the cache line is dirty (i.e., it has uncommitted data), it is written to memory before invalidation. The `clflush` instruction is also ordered by the `mfence` instruction. Therefore, to commit a series of memory writes, we first execute an `mfence` as a barrier to them, execute a `clflush` on every cacheline of all modified memory regions that need to be committed to persistent memory, and then execute another `mfence`. In this paper, we refer to this instruction sequence as a *`flush`*. As microbenchmarks in Chapter 5.2 show, using `flush` will be acceptable for most workloads.

While this description and tracking dirty memory might seem complex, this was easy to implement in practice and can be abstracted away by macros or helper functions. In particular, for data structures, all updates occur behind an API and therefore the process of `flush`ing data to non-volatile memory is hidden from the programmer. Using the simplified hash table example described above, the implementation would first write the object and `flush` it. Only after this would it write the pointer value and then `flush` again. This two-step process is transparent to the user as it occurs inside the insert method.

Finally, one should note that while `flush` is necessary for durability and consistency, it is not sufficient by itself. If any metadata update (e.g., rebalancing a tree) requires an atomic update greater than the 8 byte atomic write provided by the hardware, a failure could leave it in an inconsistent state. We therefore need the versioning approach described below in Chapters 4.2 and 4.4.

4.2 CDDS Overview

Given the challenges highlighted at the beginning of Chapter 4, an ideal data store for non-volatile memory must have the following properties:

- **Durable**: The data store should be durable. A fail-stop failure should not lose committed data.
- **Consistent**: The data store should remain consistent after every update operation. If a failure occurs during an update, the data store must be restored to a consistent state before further updates are applied.
- **Scalable**: The data store should scale to arbitrarily-large sizes. When compared to traditional data stores, any space, performance, or complexity overhead should be minimal.
- **Easy-to-Program**: Using the data store should not introduce undue complexity for programmers or unreasonable limitations to its use.

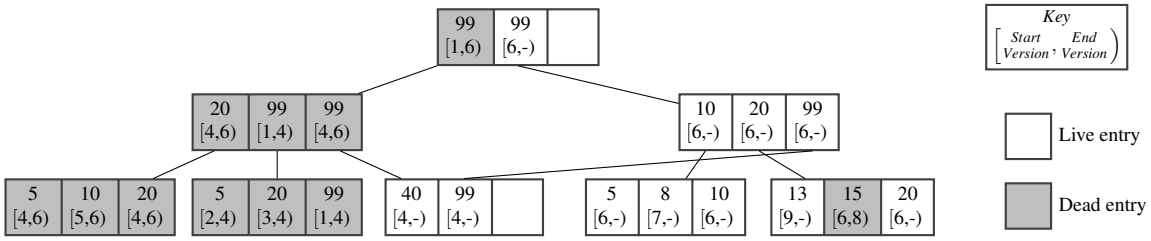


Figure 4.1: Example of a CDDS B-Tree

We believe it is possible to meet the above properties by storing data in Consistent and Durable Data Structures (CDDSs), i.e., hardened versions of conventional data structures currently used with volatile memory. The ideas used in constructing a CDDS are applicable to a wide variety of linked data structures and, in this paper, we implement a CDDS B-Tree because of its non-trivial implementation complexity and widespread use in storage systems. We would like to note that the design and implementation of a CDDS only addresses *physical* consistency, i.e., ensuring that the data structure is readable and never left in a corrupt state. Higher-level layers control *logical* consistency, i.e., ensuring that the data stored in the data structure is valid and matches external integrity constraints. Similarly, while our current system implements a simple concurrency control scheme, we do not mandate concurrency control to provide isolation as it might be more efficient to do it at a higher layer.

A CDDS is built by maintaining a limited number of versions of the data structure with the constraint that an update should not weaken the structural integrity of an older version and that updates are atomic. This versioning scheme allows a CDDS to provide consistency without the additional overhead of logging or shadowing. A CDDS thus provides a guarantee that a failure between operations will never leave the data in an inconsistent state. As a CDDS never acknowledges completion of an update without safely committing it to non-volatile memory, it also ensures that there is no silent data loss.

4.3 Versioning for Durability

Internally, a CDDS maintains the following properties:

- There exists a version number for the most recent consistent version. This is used by any thread which wishes to read from the data structure.
- Every update to the data structure results in the creation of a new version.
- During the update operation, modifications ensure that existing data contained in older versions is never over-written. Such modifications are performed by either using atomic operations or copy-on-write style changes.
- After all the modifications for an update have been made persistent, the most recent consistent version number is updated atomically and the operation is committed .

4.3.1 Garbage Collection

Along with support for multiple versions, a CDDS also tracks versions of the data structure that are being accessed. Knowing the oldest version which has a non-zero reference count has two benefits. First, we can garbage collect older versions of the data structure. Garbage collection (GC) is run in the background and helps bound the space utilized by eliminating data that will not be referenced in the future. Second, knowing the oldest active version can also improve performance by enabling intelligent space reuse in a CDDS. When creating a new entry, the CDDS can proactively reclaim the space used by older inactive versions.

4.3.2 Failure Recovery

Insert or delete operations may be interrupted due to operating system crashes or power failures. By definition, the most recent consistent version of the data structure should be accessible on recovery. However, an in-progress update needs to be removed as it belongs to an uncommitted version. We handle failures in a CDDS by using a 'forward garbage collection' procedure during recovery. This process involves discarding all update operations which were executed after the most recent consistent version. Newly created entries can be discarded while older entries with in-progress update operations are reverted.

4.4 CDDS B-Trees

As an example of a CDDS, we selected the B-Tree [18] data structure because of its widespread use in databases, file systems, and storage systems. This section discusses the design and implementation of a consistent and durable version of a B-Tree. Our B-Tree modifications[1] have been heavily inspired by previous work on multi-version data structures [6, 65]. However, our focus on durability required changes to the design and impacted our implementation. We also do not retain all previous versions of the data structure and can therefore optimize updates.

[1] In reality, our B-Tree is a B+ Tree with values only stored in leaves.

Algorithm 1: CDDS B-Tree Lookup

```
   Input: k: key, r: root
   Output: val: value
 1 begin lookup(k, r)
 2     v ← current_version
 3     n ← r
 4     while is_inner_node(n) do
 5         entry_num ← find(k, n, v)
 6         n ← n[entry_num].child
 7     entry_num ← find(k, n, v)
 8     return n[entry_num].value
 9 end
10 begin find(k, n, v)
11     l ← 0
12     h ← get_num_entries(n)
13     while l < h do                                          // Binary Serch
14         m ← (l + h)/2
15         if k ≤ n[m].key then
16             h ← m − 1
17         else l ← m + 1
18     while h < get_num_entries(n) do
19         if n[h].start ≤ v then
20             if n[h].end > v  ||  n[h].end = 0 then
21                 break
22         h ← h + 1
23     return h
24 end
```

In a CDDS B-Tree node, shown in Figure 4.1, the key and value stored in a B-Tree entry is augmented with a start and end version number, represented by unsigned 64-bit integers. A B-Tree node is considered 'live' if it has at least one live entry. In turn, an entry is considered 'live' if it does not have an end version (displayed as a '$-$' in Figure 4.1). To bound space utilization, in addition to ensuring that a minimum number of entries in a B-Tree node are used, we also bound the minimum number of live entries in each node. Thus, while the CDDS B-Tree API is identical to normal B-Trees, the implementation differs significantly. In the rest of this section, we use the *lookup*, *insert*, and *delete* operations to illustrate how the CDDS B-Tree design guarantees consistency and durability.

4.4.1 Lookup

We first briefly describe the lookup algorithm, shown in Algorithm 1. For ease of explanation, the pseudocode in this and following algorithms does not include all of the design details. The algorithm uses the `find` function to recurse down the tree (lines 4–6) until it finds the leaf node with the correct key and value.

Consider a lookup for the key 10 in the CDDS B-Tree shown in Figure 4.1. After determining the most current version (version 9, line 2), we start from the root node and pick the rightmost entry with key 99 as it is the next largest

Algorithm 2: CDDS B-Tree Insertion

```
   Input: k: key, r: root
 1 begin insert_key(k, r)
 2     v ← current_version
 3     v' ← v + 1
       // Recurse to leaf node (n)
 4     y ← get_num_entries(n)
 5     if y = node_size then                                          // Node Full
 6         if entry_num = can_reuse_version(n, y) then
               // Remove the oldest entry
 7             n[entry_num].key ← k
 8             n[entry_num].start ← v'
 9             n[entry_num].end ← 0
10             flush(n[entry_num])
11         else
12             split_insert(n, k, v')
               // Update inner nodes
13     else
14         n[y].key ← k
15         n[y].start ← v'
16         n[y].end ← 0
17         flush(n[y])
18     current_version ← v'
19     flush(current_version)
20 end
21 begin can_reuse_version(n, y)
22     cv ← get_last_consistent_version
23     entry_num ← 0
24     for i = 1 to y do
25         if n[i].end > 0 and n[i].end < cv then
26             if entry_num = 0 or i < entry_num then
27                 entry_num = i
28     return entry_num
29 end
```

valid key. Similarly in the next level, we follow the link from the leftmost entry and finally retrieve the value for 10 from the leaf node.

Our implementation currently optimizes lookup performance by ordering node entries by key first and then by the start version number. This involves extra writes during inserts to shift entries but improves read performance by enabling a binary search within nodes (lines 13–17 in `find`). While we have an alternate implementation that optimizes writes by not ordering keys at the cost of higher lookup latencies, we do not use it as our target workloads are read-intensive. Finally, once we detect the right index in the node, we ensure that we are returning a version that was valid for v, the requested version number (lines 18–22).

Algorithm 3: CDDS B-Tree Split-Insert

```
   Input: n: node, k: key, version: v
 1 begin split_insert(n, k, v)
 2     l ← num_live_entries(n)
 3     m_l ← min_live_entries
 4     if l > 4 * m_l then
 5         nn_1 ← new_node
 6         nn_2 ← new_node
 7         for i = 1 to l/2 do
 8             insert(nn_1, n[i].key, v)
 9         for i = l/2 + 1 to l do
10             insert(nn_2, n[i].key, v)
11         if k < n[l/2].key then
12             insert(nn_1, k, v)
13         else insert(nn_2, k, v)
14         flush(nn_1, nn_2)
15     else
16         nn ← new_node
17         for i = 1 to l do
18             insert(nn, n[i].key, v)
19         insert(nn, k, v)
20         flush(nn)
21     for i = 1 to l do
22         n[i].end ← v
23     flush(n)
24 end
```

4.4.2 Insertion

The algorithm for inserting a key into a CDDS B-Tree is shown in Algorithm 2. Our implementation of the algorithm uses the `flush` operation (described in Chapter 4.1) to perform atomic operations on a cacheline. Consider the case where a key, 12, is inserted into the B-Tree shown in Figure 4.1. First, an algorithm similar to lookup is used to find the leaf node that contains the key range that 12 belongs to. In this case, the right-most leaf node is selected. As shown in lines 2–3, the current consistent version is read and a new version number is generated. As the leaf node is full, we first use the `can_reuse_version` function to check if an existing dead entry can be reused. As shown in line 25, an entry can be reused if its end version is older than the current consistent version. In this case, the entry with key 15 died at version 8 and is reused. To reuse a slot we first remove the key from the node and shift the entries to maintain them in sorted order. Now we insert the new key and again shift entries as required. For each key shift, we ensure that the data is first `flush`ed to another slot before it is overwritten. This ensures that the safety properties specified in Chapter 4.3 are not violated. While not described in the algorithm, if an empty entry was detected in the node, it would be used and the order of the keys, as specified in Chapter 4.4.1, would be maintained.

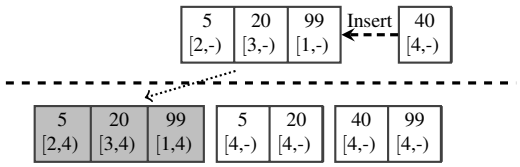


Figure 4.2: CDDS node split during insertion

If no free or dead entry was found, a `split_insert`, similar to a traditional B-Tree split, would be performed. `split_insert`, shown in Algorithm 3, is a copy-on-write style operation in which existing entries are copied before making a modification. As an example, consider the node shown in Figure 4.2, where the key 40 is being inserted. We only need to preserve the 'live' entries for further updates and `split_insert` creates one or two new nodes based on the number of live entries present. Note that setting the end version (lines 21–22) is the only change made to the existing leaf node. This ensures that older data versions are not affected by failures. In this case, two new nodes are created at the end of the split.

We perform two operations to update the inner nodes after such a split. First, the entry corresponding to the now-dead child node is marked as dead. Next, new entries are inserted with links pointing to the newly created leaf nodes. If the inner node overflows during this process a `split_insert` is performed to create new inner nodes. When the root node of a tree overflows, we split the root node and create one or two new nodes. We then create a new root node with links to the old root and to the newly created split-nodes. The pointer to the root node is updated atomically to ensure safety.

Once all the changes have been `flush`ed to persistent storage, the current consistent version is updated atomically (lines 18–19). At this point, the update has been successfully committed to the NVBM and failures will not result in the update being lost.

4.4.3 Deletion

Deleting an entry is conceptually simple as it simply involves setting the end version number for the given key. It does not require deleting any data as that is handled by GC. However, in order to bound the number of live blocks in the B-Tree and improve space utilization, we shift live entries if the number of live entries per node reaches m_l, a threshold defined in Chapter 4.4.6. The only exception is the root node as, due to a lack of siblings, shifting within the same level is not feasible. However, as described in Chapter 4.4.4, if the root only contains one live entry, the child will be promoted.

As shown in Algorithm 4, we first check if the sibling has at least $3 \times m_l$ live entries and, if so, we copy m_l live entries from the sibling to form a new node. As the leaf has m_l live entries, the new node will have $2 \times m_l$ live entries.

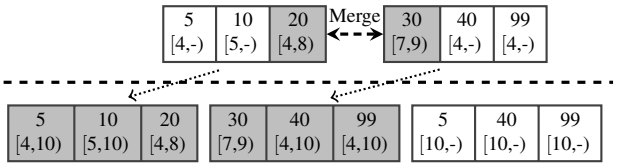


Figure 4.3: CDDS node merge during deletion

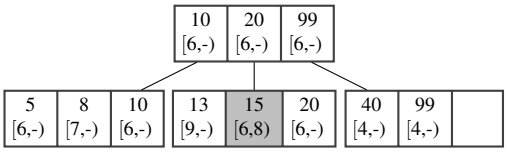


Figure 4.4: CDDS B-Tree after Garbage Collection

If that is not the case, we check if the sibling has enough space to copy the live entries. Otherwise, as shown in Figure 4.3, we merge the two nodes to create a new node containing the live entries from the leaf and sibling nodes. The number of live entries in the new node will be $\geq 2 \times m_l$. The copy and merge operations only update the end versions of existing entries and do not affect their contents. The inner nodes are updated by marking end versions for the now-dead leaf nodes and adding pointers to any newly created nodes. Since new nodes are inserted in this operation, the inner nodes could overflow and we use functions described in Chapter 4.4.2 to handle this. After all the changes have been flushed to persistent memory, the current consistent version is incremented.

4.4.4 Garbage Collection

As shown in Chapter 4.4.3, the size of the B-Tree does not decrease when keys are deleted and can increase due to the creation of new nodes. To reduce the space overhead, we therefore use a periodic GC procedure. It should be noted that while we use versioning as technique for performing consistent updates, we do not need to retain obsolete versions.

The GC procedure needs to take into account the fact that a B-Tree node can have more than one parent. This can happen during the `split_insert` operation and an example can be seen in Figure 4.1. For this reason, we divide the GC procedure into two parts. The first is a *logical* procedure which removes pointers that will no longer be used and the second, a *physical* procedure where dead nodes are deleted and allocated memory is reclaimed. The former procedure is described in Algorithm 5 and starts from the root of the B-Tree. If an inner node entry has died before the current consistent version, the pointer from the inner node to the child node can be removed. This is because the pointer is only used to lookup elements older than the current consistent version and will not be used by future operations. Any live entries are preserved and entries are left-shifted to overwrite removed entries. This procedure is recursively repeated till we reach the leaf nodes. If a node contains only one live entry after garbage collection, the child pointed to by the entry is promoted. This helps reduce the height of the B-Tree. As seen in the transformation of Figure 4.1 to the reduced-height tree shown in Figure 4.4, only live nodes are present after GC. *Physical* GC is currently implemented using a mark-and-sweep garbage collector [10]. While a reference counting implementation might also suffice, it would entail the overhead of flushing reference counts.

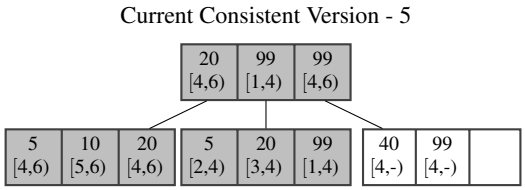


Figure 4.5: CDDS B-Tree before Recovery

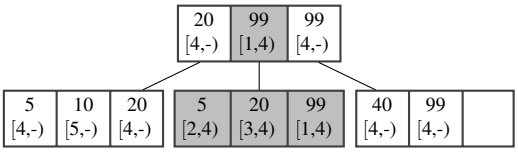


Figure 4.6: CDDS B-Tree after Recovery

4.4.5 Failure Recovery

The recovery procedure for the B-Tree is similar to garbage collection. In Algorithm 6, we describe the logical phase in which we remove entries newer than the current consistent version. If an inner node entry was created after the current consistent version, the pointer to the child node can be removed as the update was interrupted before completion. Also, if an entry was deleted after the current consistent version, then the entry is marked as live. This procedure is repeated recursively till we reach the leaf nodes. Additionally, if there was a failure while inserting in a sorted order, we could have duplicate entries in a node. We remove duplicate entries by shifting entries to the left and use the `flush` operation to ensure we don't introduce any inconsistencies during recovery.

For example, consider the CDDS B-Tree shown in Figure 4.5. The update operation which inserts version 6 has been interrupted due to a failure. The end versions for nodes are marked as 6, but newly created nodes have not been linked to the B-Tree before the failure. Also, the current consistent version for the B-Tree is known to be 5. In this case, the recovery function changes the entries which ended at version 6 to be live as shown in Figure 4.6. Hence, the recovery function performs a physical 'undo' of partial updates and ensures that the tree is physically and logically identical to the most recent consistent version. While our current recovery implementation scans the entire data structure, the recovery process is fast as it operates at memory bandwidth speeds and only needs to verify CDDS metadata. While we believe that the recovery procedure is re-entrant, we are working towards verifying it using empirical evidence from fault injection tests. We are also exploring formal verification techniques that can be used to verify the recovery protocol used in the CDDS B-Tree.

4.4.6 Space Analysis

In the CDDS B-Tree, space utilization can be characterized by the number of live blocks required to store N key-value pairs. Since the values are only stored in the leaf nodes, we analyze the maximum number of live leaf nodes present in the tree. In the CDDS B-Tree, a new node is created by an insert or delete operation. As described in Chapters 4.4.2 and 4.4.3, the minimum number of live entries in new nodes is $2 \times m_l$.

When the number of live entries in a node reaches m_l, it is either merged with a sibling node or its live entries are copied to a new node. Hence, the number of live entries in a node is $> m_l$. Therefore, in a B-Tree with N live keys,

the maximum number of live leaf nodes is bound by $O(\frac{N}{m_l})$. Choosing m_l as $\frac{k}{5}$, where k is the size of a B-Tree node, the maximum number of live leaf nodes is $O(\frac{5N}{k})$.

For each live leaf node, there is a corresponding entry in the parent node. Since the number of live entries in an inner node is also $> m_l$, the number of parent nodes required is $O\left(\frac{\frac{5N}{k}}{m_l}\right) = O(\frac{N}{(\frac{k}{5})^2})$. Extending this, we can see that the height of the CDDS B-Tree is bound by $O(log_{\frac{k}{5}}N)$. This also bounds the time for all B-Tree operations.

4.5 CDDS Discussion

Apart from the CDDS B-Tree operations described above, the implementation also supports additional features including iterators and range scans. We believe that CDDS versioning also lends itself to other powerful features such as instant snapshots, rollback for programmer recovery, and integrated NVBM wear-leveling. We hope to explore these issues in our future work.

One of the limitations of our CDDS-BTree implementation is that we have an equal number of pointers and keys in an inner node. This is different from a traditional B-Tree which has an extra pointer for keys larger than those stored in the node. This is an implementation artifact that makes it easier to track the start and end versions of a pointer. As a consequence, the key with maximum value needs to be first inserted to avoid any imbalance while inserting other keys. In practice this is overcome easily by having a maximum key which the comparator function recognizes and this imposes limited overhead in the implementation.

We also do not anticipate the design of a CDDS preventing the implementation of different concurrency schemes. Our current CDDS B-Tree implementation uses a multiple-reader, single-writer model. However, the use of versioning lends itself to more complex concurrency control schemes including multi-version concurrency control (MVCC) [8]. While beyond the scope of this paper, exploring different concurrency control schemes for CDDSs is a part of our future work.

Finally, apart from multi-version data structures [6, 65], CDDSs have also been influenced by Persistent Data Structures (PDSs) [23]. The "Persistent" in PDS does not actually denote durability on persistent storage but, instead, represents immutable data structures where an update always yields a new data structure copy and never modifies previous versions. The CDDS B-Tree presented above is a weakened form of semi-persistent data structures. We modify previous versions of the data structure for efficiency but are guaranteed to recover from failure and rollback to a consistent state. However, the PDS concepts are applicable, in theory, to all linked data structures. Using PDS-style techniques, we have implemented a proof-of-concept CDDS hash table and, as evidenced by previous work for functional programming languages [46], we are confident that CDDS versioning techniques can be extended to a wide range of data structures.

4.6 Tembo: A CDDS Key-Value Store

We created Tembo, a CDDS Key-Value (KV) store, to evaluate the effectiveness of a CDDS-based data store. The system involves the integration of the CDDS-based B-Tree described in Chapter 4.4 into Redis [53], a widely used event-driven KV store. As our contribution is not based around the design of this KV system, we only briefly describe Tembo in this section. As shown in Chapter 5.4, the integration effort was minor and leads us to believe that retrofitting CDDS into existing applications will be straightforward.

The base architecture of Redis is well suited for a CDDS as it retains the entire data set in RAM. This also allows an unmodified Redis to serve as an appropriate performance baseline. While persistence in the original system was provided by a write-ahead append-only log, this is eliminated in Tembo because of the CDDS B-Tree integration. For fault-tolerance, Tembo provides master-slave replication with support for hierarchical replication trees where a slave can act as the master for other replicas. Consistent hashing [36] is used by client libraries to distribute data in a Tembo cluster.

Algorithm 4: CDDS B-Tree Deletion

```
   Input: k: key, r: root
 1 begin delete(k, r)
 2     v ← current_version
 3     v' ← v + 1
       // Recurse to leaf node (n)
 4     y ← find_entry(n, k)
 5     n[y].end ← v'
 6     l ← num_live_entries(n)
 7     if l = m_l then                                              // Underflow
 8         s ← pick_sibling(n)
 9         l_s ← num_live_entries(s)
10         if l_s > 3 × m_l then
11             copy_from_sibling(n, s, v')
12         else merge_with_sibling(n, s, v')
           // Update inner nodes
13     else flush(n[y])
14     current_version ← v'
15     flush(current_version)
16 end
   Input: n: node, s: sibling, v:version
17 begin merge_with_sibling(n, s, v)
18     y ← get_num_entries(s)
19     if y < 4 × m_l then
20         for i = 1 to m_l do
21             insert(s, n[i].key, v)
22             n[i].end ← v
23     else
24         nn ← new_node
25         l_s ← num_live_entries(s)
26         for i = 1 to l_s do
27             insert(nn, s[i].key, v)
28             s[i].end ← v
29         for i = 1 to m_l do
30             insert(nn, n[i].key, v)
31             n[i].end ← v
32         flush(nn)
33     flush(n, s)
34 end
   Input: n: node, s: sibling, v: version
35 begin copy_from_sibling(n, s, v)
36     nn ← new_node
37     for i = 1 to m_l do
38         insert(nn, s[i].key)
39         s[i].end ← v
40         insert(nn, n[i].key)
41         n[i].end ← v
42     flush(nn, n, s)
43 end
```

Algorithm 5: CDDS B-Tree Garbage Collection

```
   Input: n: node
1  begin garbage_collect(n)
2      if is_inner_node(n) then
3          v ← get_last_consistent_version
4          shift ← 0
5          y ← get_num_entries(n)
6          for i = 1 to y do
7              if n[i].end ≠ 0 and n[i].end < v then
8                  n[i].child = null
9                  shift++
10             else
11                 garbage_collect(n[i].child)
12                 if shift ≠ 0 then
13                     n[i − shift].child = n[i].child
14                     n[i − shift].key = n[i].key
15                     n[i − shift].start = n[i].start
16                     n[i − shift].end = n[i].end
17         set_num_entries(n, y − shift)
18         if get_num_entries(n) = 1 then
               // Use only child instead
19 end
```

Algorithm 6: CDDS B-Tree Recovery

```
   Input: n: node
1  begin recover_node(n)
2      v ← get_last_consistent_version
3      shift ← 0
4      y ← get_num_entries(n)
5      for i = 1 to y do
6          if n[i].start > v then
               // Created during
               // partial operation
7              shift++
8          else
9              if n[i].end > v then
                   // Deleted during
                   // partial operation
10                 n[i].end = 0
11             if is_inner_node(n) then
12                 recover_node(n[i].child)
13             if shift ≠ 0 then
14                 n[i − shift].key = n[i].key
15                 n[i − shift].start = n[i].start
16                 n[i − shift].end = n[i].end
17     set_num_entries(n, y − shift)
18 end
```

Chapter 5

Evaluation

In this section, we evaluate our design choices in building Consistent and Durable Data Structures. First, we measure the overhead associated with techniques used to achieve durability on existing processors. We then compare the CDDS B-tree to Berkeley DB and against log-based schemes. After briefly discussing CDDS implementation and integration complexity, we present results from a multi-node distributed experiment where we use the Yahoo Cloud Serving Benchmark (YCSB) [20].

5.1 Evaluation Setup

As NVBM is not commercially available yet, we used DRAM-based servers. While others [19] have shown that DRAM-based results are a good predictor of NVBM performance, as a part of our ongoing work, we aim to run micro-architectural simulations to confirm this within the context of our work. Our testbed consisted of 15 servers with two Intel Xeon Quad-Core 2.67 GHz (X5550) processors and 48 GB RAM each. The machines were connected via a full-bisection Gigabit Ethernet network. Each processor has 128 KB L1, 256 KB L2, and 8 MB L3 caches. While each server contained 8 300 GB 10K SAS drives, unless specified, all experiments were run directly on RAM or on a ramdisk. We used the Ubuntu 10.04 Linux distribution and the 2.6.32-24 64-bit kernel.

5.2 Flush Performance

To accurately capture the performance of the `flush` operation defined in Chapter 4.1, we used the "MultCallFlushLRU" methodology [67]. The experiment allocates 64 MB of memory and subdivides it into equally-sized cache-aligned objects. Object sizes ranged from 64 bytes to 64 MB. We write to every cache line in an object, `flush` the entire object, and then repeat the process with the next object. For improved timing accuracy, we stride over the memory region multiple times.

Remembering that each `flush` is a number of `clflush`es bracketed by `mfence`s on both sides, Figure 5.1 shows the number of `clflush`es executed per second. Flushing small objects sees the worst performance ($\sim$12M

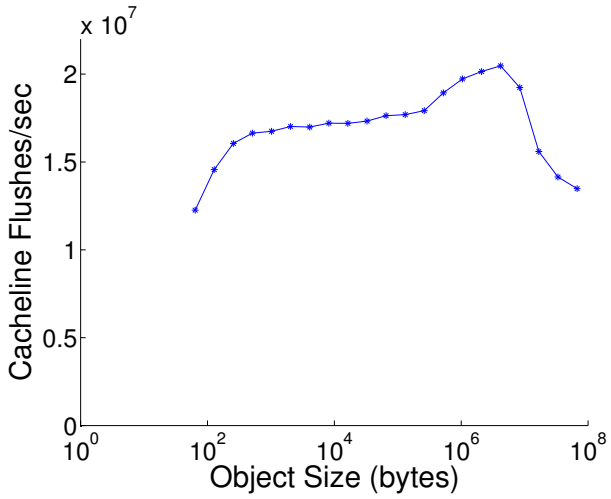


Figure 5.1: Flushes/second

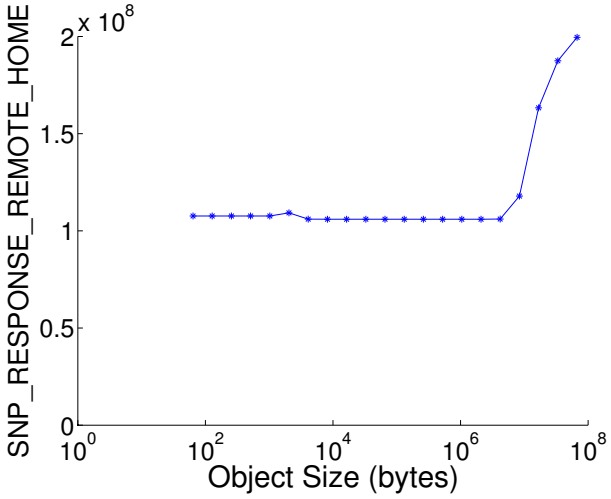


Figure 5.2: Cache Snooping

cacheline flushes/sec for 64 byte objects). For larger objects (256 bytes–8 MB), the performance ranges from ∼16M–20M cacheline flushes/sec.

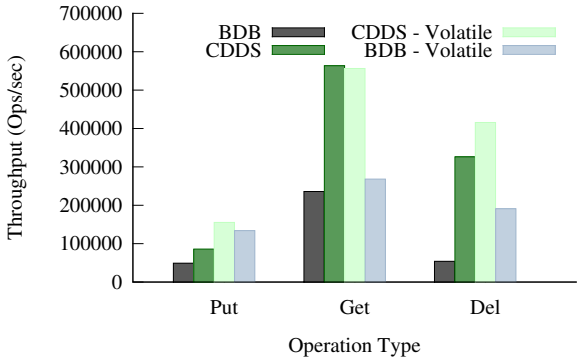
We also observed an unexpected drop in performance for large objects (>8 MB). Our analysis showed that this was due to the cache coherency protocol. Large objects are likely to be evicted from the L3 cache before they are explicitly flushed. A subsequent `clflush` would miss in the local cache and cause a high-latency "snoop" request that checks the second off-socket processor for the given cache line. As measured by the UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE performance counter, seen in Figure 5.2, the second socket shows a corresponding spike in requests for cache lines that it does not contain. To verify this, we physically removed a processor and observed that the anomaly disappeared[1]. Further, as we could not replicate this slowdown on AMD platforms, we believe that cache-coherency protocol modifications can address this anomaly.

Overall, the results show that we can `flush` 0.72–1.19 GB/s on current processors. For applications without networking, Chapter 5.3 shows that future hardware support can help but applications using `flush` can still outperform applications that use file system `sync` calls. Distributed applications are more likely to encounter network bottlenecks before `flush` becomes an overhead.

5.3 API Microbenchmarks

This section compares the CDDS B-Tree performance for puts, gets, and deletes to Berkeley DB's (BDB) B-Tree implementation [47]. For this experiment, we insert, fetch, and then delete 1 million key-value tuples into each system. After each operation, we flush the CPU cache to eliminate any variance due to cache contents. Keys and

[1]We did not have physical access to the experimental testbed and ran the processor removal experiment on a different dual-socket Intel Xeon (X5570) machine.



Mean of 5 trials. Max. standard deviation: 2.2% of the mean.

Figure 5.3: Berkeley DB Comparison

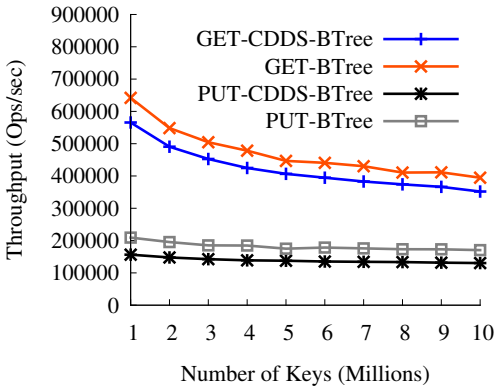


Figure 5.4: Versioning Overhead

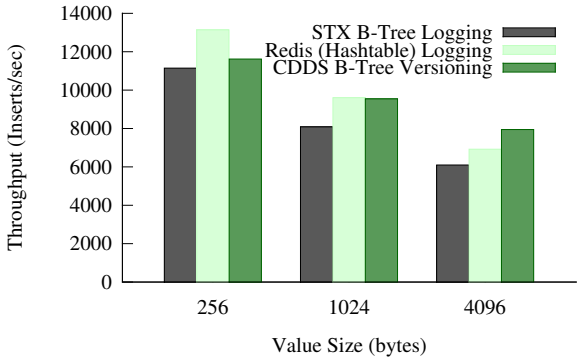
values are 25 and 2048 bytes large. The single-threaded benchmark driver runs in the same address space as BDB and CDDS. BDB's cache size was set to 8 GB and could hold the entire data set in memory. Further, we configure BDB to maintain its log files on an in-memory partition.

We run both CDDS and BDB (v4.8) in durable and volatile modes. For BDB volatile mode, we turn transactions and logging off. For CDDS volatile mode, we turn `flush`ing off. Both systems in volatile mode can lose or corrupt data and would not be used where durability is required. We only present the volatile results to highlight predicted performance if hardware support was available and to discuss CDDS design tradeoffs.

The results, displayed in Figure 5.3, show that, for memory-backed BDB in durable mode, the CDDS B-Tree improves throughout by 74%, 138%, and 503% for puts, gets, and deletes respectively. These gains come from not using a log (extra writes) or the file system interface (system call overhead). CDDS delete improvement is larger than puts and gets because we do not delete data immediately but simply mark it as dead and use GC to free unreferenced memory. In results not presented here, reducing the value size, and therefore the log size, improves BDB performance but CDDS always performs better.

If zero-overhead epoch-based hardware support [19] was available, the CDDS volatile numbers show that performance of puts and deletes would increase by 80% and 27% as `flush`es would never be on the critical path. We do not observe any significant change for gets as the only difference between the volatile and durable CDDS is that the `flush` operations are converted into a noop.

We also notice that while volatile BDB throughput is lower than durable CDDS for gets and dels by 52% and 41%, it is higher by 56% for puts. Puts are slower for the CDDS B-Tree because of the work required to maintain key ordering (described in Chapter 4.4.1), GC overhead, and a slightly higher height due to nodes with a mixture of live and dead entries. Volatile BDB throughput is also higher than durable BDB but lower than volatile CDDS for all operations.



Mean of 5 trials. Max. standard deviation: 6.7% of the mean.

Figure 5.5: Versioning vs. Logging

	Lines of Code
Original STX B-Tree	2,110
CDDS Modifications	1,902
Redis (v2.0.0-rc4)	18,539
Tembo Modifications	321

Table 5.1: Lines of Code Modified

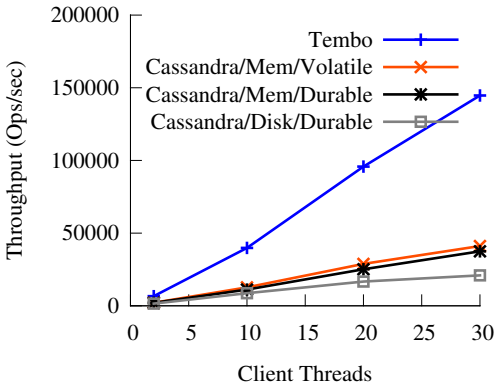
To measure versioning overhead, we compared the volatile CDDS B-Tree to a normal B-Tree [9]. We performed the same experiment described above and varied the input from one to ten million keys. From the results, shown in Figure 5.4, we can see that the performance of the CDDS-BTree remains consistent as we increase the number of keys. Also, we can see that volatile CDDS's performance was lower than the in-memory B-Tree by 21%-25% for puts and by 9%-12% for gets. This difference is similar to other performance-optimized versioned B-trees [58].

5.4 Implementation Effort

The CDDS B-Tree started with the STX C++ B-Tree [9] implementation but, as measured by `sloccount` and shown in Table 5.1, the addition of versioning and NVBM durability replaced 90% of the code. While the API remained the same, the internal implementation differs substantially. The integration with Redis to create Tembo was simpler and only changed 1.7% of code and took less than a day to integrate. Since the CDDS B-Tree implements an interface similar to an STL Sorted Container, we believe that integration with other systems should also be simple. Overall, our experiences show that while the initial implementation complexity is moderately high, this only needs to be done once for a given data structure. The subsequent integration into legacy or new systems is straightforward.

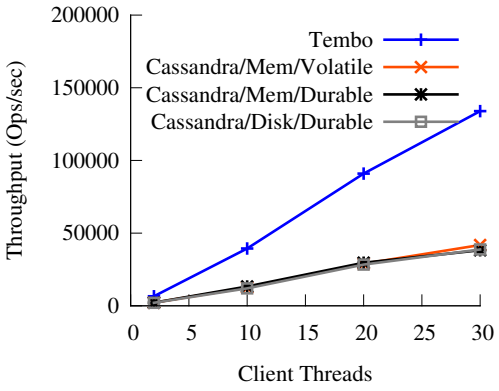
5.5 Tembo Versioning vs. Redis Logging

Apart from the B-Tree specific logging performed by BDB in Chapter 5.3, we also wanted to compare CDDS versioning when integrated into Tembo to the write-ahead log used by Redis in fully-durable mode. Redis uses a hashtable and, as it is hard to compare hashtables and tree-based data structures, we also replaced the hashtable with the STX



Mean of 5 trials. Max. standard deviation: 7.8% of the mean.

Figure 5.6: YCSB: SessionStore



Mean of 5 trials. Max. standard deviation: 8.1% of the mean.

Figure 5.7: YCSB: StatusUpdates

B-Tree. In this single-node experiment, we used 6 Tembo or Redis data stores and 2 clients[2]. The write-ahead log for the Redis server was stored on an in-memory partition mounted as `tmpfs` and did not use the hard disk. Each client performed 1M inserts over the loopback interface.

The results, presented in Figure 5.5, show that as the value size is increased, Tembo performs up to 30% better than Redis integrated with the STX B-Tree. While Redis updates the in-memory data copy and also writes to the append-only log, Tembo only updates a single copy. While hashtable-based Redis is faster than Tembo for 256 byte values because of faster lookups, even with the disadvantage of a tree-based structure, Tembo's performance is almost equivalent for 1 KB values and is 15% faster for 4 KB values.

The improvements for the experiments in this section are lower than those in Chapter 5.3 because of network latency overhead. The `fsync` implementation in `tmpfs` also does not explicitly flush modified cache lines to memory and is therefore biased against Tembo. We are working on modifications to the file system that will enable a fairer comparison. Finally, some of the overhead is due to maintaining ordering properties in the CDDS-based B-Tree to support range scans - a feature not used in the current implementation of Tembo.

5.6 End-to-End Comparison

For an end-to-end test, we used YCSB, a framework for evaluating the performance of Key-Value, NoSQL, and cloud storage systems [20]. In this experiment, we used 13 servers for the cluster and 2 servers as the clients. We extended YCSB to support Tembo, and present results from two of YCSB's workloads. Workload-A, referred to as SessionStore in this section, contains a 50:50 read:update mix and is representative of tracking recent actions in an online user's session. Workload-D, referred to as StatusUpdates, has a 95:5 read:insert mix. It represents people updating their

[2]Being event-driven, both Redis and Tembo are single-threaded. Therefore one data store (or client) is run per core in this experiment.

online status (e.g., Twitter tweets or Facebook wall updates) and other users reading them. Both workloads execute 2M operations on values consisting of 10 columns with 100 byte fields.

We compare Tembo to Cassandra (v0.6.1) [38], a distributed data store that borrows concepts from BigTable [14] and Dynamo [22]. We used three different Cassandra configurations in this experiment. The first two used a ramdisk for storage but the first (Cassandra/Mem/Durable) flushed its commit log before every update while the second (Cassandra/Mem/Volatile) only flushed the log every 10 seconds. For completeness, we also configured Cassandra to use a disk as the backing store (Cassandra/Disk/Durable).

Figure 5.6 presents the aggregate throughput for the SessionStore benchmark. With 30 client threads, Tembo's throughput was 286% higher than memory-backed durable Cassandra. Given Tembo and Cassandra's different design and implementation choices, the experiment shows the overheads of Cassandra's in-memory "memtables," on-disk "SSTables," and a write-ahead log, vs. Tembo's single-level store. Disk-backed Cassandra's throughput was only 22–44% lower than the memory-backed durable configuration. The large number of disks in our experimental setup and a 512 MB battery-backed disk controller cache were responsible for this better-than-expected disk performance. On a different machine with fewer disks and a smaller controller cache, disk-backed Cassandra bottlenecked with 10 client threads. Formal performance models [64] designed for disk arrays have also shown the benefits of using battery-backed disk controller caches.

Figure 5.7 shows that, for the StatusUpdates workload, Tembo's throughput is up to 250% higher than memory-backed durable Cassandra. Tembo's improvement is slightly lower than the SessionStore benchmark because StatusUpdates insert operations update all 10 columns for each value, while the SessionStore only selects one random column to update. Finally, as the entire data set can be cached in memory and inserts represent only 5% of this workload, the different Cassandra configurations have similar performance.

Chapter 6

Profiling and Analysis

In order to compare different design choices and get a deeper understanding of CDDS, we perform experiments using Cafegrind [13], an extension to the Valgrind [45] memory profiling framework. The profiler instruments all memory accesses and memory allocations and gives us an accurate count of the number and frequency of memory operations to heap allocations. Moreover, since Cafegrind can also infer the types of heap allocations, we can restrict the analysis to objects that we believe will be stored in NVBM. By measuring the number of writes to NVBM, we can estimate the bandwidth usage and power consumption for different configurations. Furthermore, knowing which NVBM locations are updated frequently can help design wear-leveling algorithms.

6.1 Write Frequency by Location

In order to measure how CDDS-versioning can help in wear-leveling, we profile the number of writes made to a CDDS B-Tree and the STX B-Tree. We run the same benchmark used in Section 5.3 but insert only 1000 keys to help visualize the number of writes per B-Tree node. The keys are chosen randomly to ensure that they are evenly spread out over the key space and repeated runs of the experiment produced similar results. Using the type information inferred by Cafegrind, we filter the store instructions which are made on B-Tree nodes during execution. As seen in Figure 6.1, Figure 6.2 the number of writes per B-Tree node is distributed in the case of a CDDS B-Tree and there are no hot spots. The STX B-Tree however has some hot spots which have many more writes than other locations. It can also be seen that there are 93 B-Tree nodes created in the CDDS B-Tree while only 47 are used in the STX B-Tree. This includes leaf nodes and inner nodes created while inserting elements in the workload.

We repeated the same experiment for a larger number of keys to see how the number of writes per page varied across a larger number of pages. The results from an experiment where 100K keys were inserted is shown in Figure 6.3. In this graph, the number of writes made to a particular page is plotted for every page used by the application. Similar to the previous figure, we can see that writes to a CDDS B-Tree are evenly distributed across a larger number of pages. We can also see that in the STX B-Tree there are some pages which have up to 16,817 writes, while other pages have

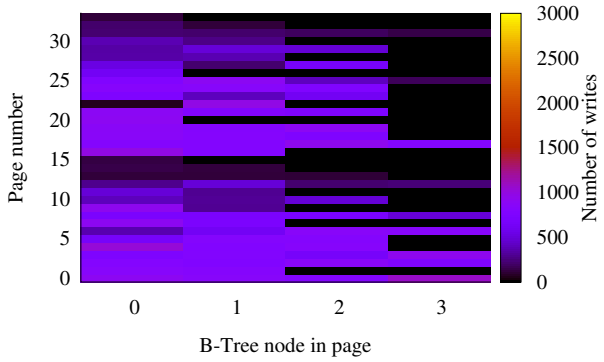


Figure 6.1: Heat map of writes in CDDS B-Tree

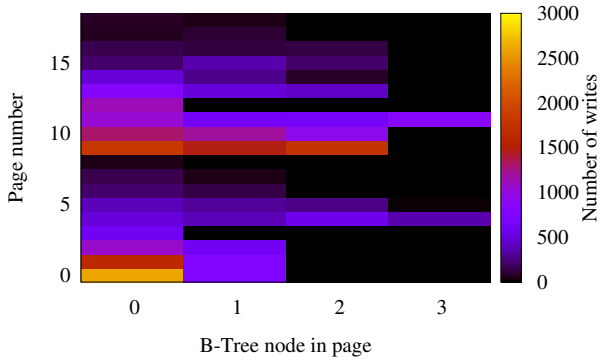


Figure 6.2: Heat map of writes in STX B-Tree

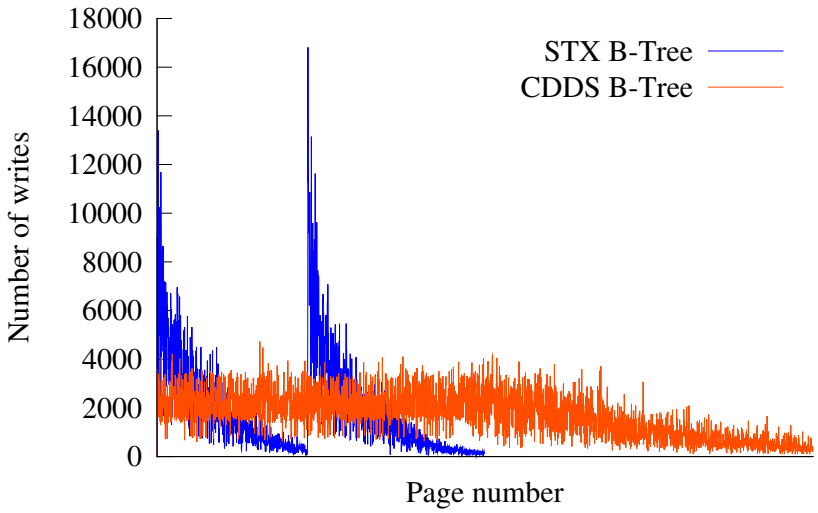


Figure 6.3: Number of writes per page in CDDS B-Tree and STX B-Tree

as few as 36 writes during the experiment. Hence we believe that CDDS-based data structures can be a useful building block to design integrated wear-leveling solutions for NVBM.

6.2 Write Frequency by Time

Profiling the CDDS B-Tree using Cafegrind also helps us understand how many writes are sent to NVBM over time. We can also check if there are some events which cause a sudden increase in the number of writes. Running the same micro-benchmark, described in the previous section, we plot the number of writes that happen with respect to time in CPU clock cycles. In figure 6.4, writes to B-Tree nodes are categorized into bins, where a bin corresponds to 2000 CPU cycles. We also plot the allocation of new B-Tree nodes in the same figure. We can see that when new nodes are created due to a `split_insert`, a large number of writes are made to NVBM.

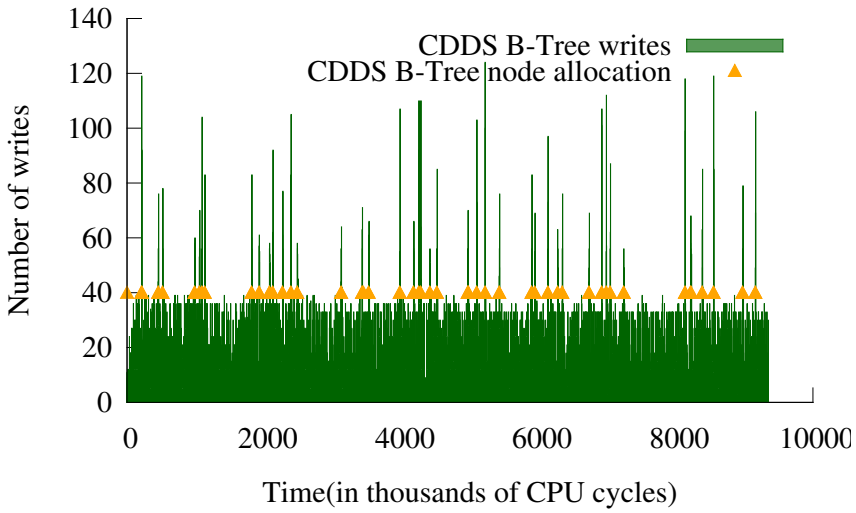


Figure 6.4: Writes and Memory allocation events in CDDS B-Tree

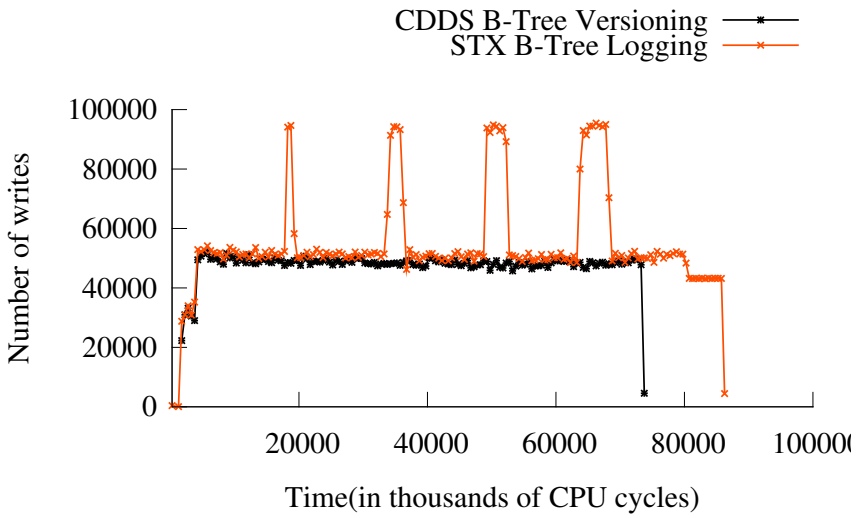


Figure 6.5: Writes for Versioning vs. Logging

6.3 Profiling Versioning vs. Logging

To compare the performance of the versioning scheme used in Tembo against the write-ahead logging scheme used in Redis we compared the throughput for both configurations in Section 5.5. Using Cafegrind, we can gain further insight into the memory operations performed in both cases and analyze how they affect the throughput. We repeated the experiment performed in Section 5.5, but used 10000 inserts because of the overhead imposed by profiling. The results, shown in Figure 6.5 show that the versioning scheme used by Tembo has fewer writes compared to the write-ahead logging scheme. We also found that the periodic extra writes in the write-ahead logging scheme were due to log

compaction operations. Finally we can see that the CDDS-based experiment finishes earlier than the logging-based experiment, confirming our earlier result in which the Tembo had a higher throughput compared to Redis.

6.4 Discussion

Although memory profiling does help us understand the behavior of CDDS, there are some limitations to this approach. Tracing writes to NVBM locations does not take into account the multi-level cache design in a processor. Also, the memory allocator and wear-leveling algorithms can change the write frequency from an application to NVBM. However, given that NVBM is not yet commercially available, we believe that memory profiling is a useful approach to compare different software designs. Memory profilers can also be used to analyze how software systems are affected by asymmetric read/write latencies of NVBM [51].

Chapter 7

Conclusion and Future Work

There are several directions in which this work can be extended to provide better support for the integration of NVBM.

Reliability

In current systems, file systems play an important role in ensuring that data is not corrupted by device failures. In CDDS-based systems, the OS (along with userspace libraries) needs to handle failures in NVBM devices. Though NVBM technologies are projected to have greater endurance than Flash-based SSDs [70], the OS will also need to provide support for wear-leveling algorithms in the persistent virtual memory subsystem. We are investigating how the integration of wear-leveling with CDDS versioning can lead to benefits in performance and reliability.

Multicore Architectures

The advent of NVBM as a replacement for disks is one among the many architectural trends, which are influencing the design and implementation of next-generation operating systems. Trends in architecture also suggest that many-core machines with tens to hundreds of cores will be available soon [2]. While operating systems like Barrelfish [5] propose OS designs where message passing is used to communicate between multiple cores, CDDS based systems propose that NVBM is made directly available to applications. It remains to be seen whether NVBM allocation and garbage collection schemes would affect operating systems [11] designed for such many-core machines.

Safety

CDDS-based systems currently depend on virtual memory mechanisms to provide fault-isolation and like other services, it depends on the OS for safety. Therefore, while unlikely, placing NVBM on the memory bus can expose it to accidental writes from rogue DMAs. In contrast, the narrow traditional block device interface makes it harder to accidentally corrupt data. We believe that hardware memory protection, similar to IOMMUs, will be required to address this problem. Given that we map data into an application's address space, stray writes from a buggy application could also destroy data. While this is no different from current applications that `mmap` their data, we are developing lightweight persistent heaps that use virtual memory protection with a RVM-style API [56] to provide improved data safety.

The end-to-end argument [55] in system design states that functionality provided at lower layers is redundant when compared to the cost of providing them. In the context of the persistent storage on NVBM, this work has shown

that low-level abstractions like the block device layer and file interface are too expensive and that we need to rethink the design of storage systems for NVBM. This work has presented the design and implementation of Consistent and Durable Data Structures (CDDSs), an architecture that, without processor modifications, allows for the creation of log-less storage systems on NVBM. Results from our experiments show that redesigning systems to support single-level data stores will be critical in meeting the high-throughput requirements of emerging applications.

References

[1] The data deluge. *The Economist*, 394(8671):11, Feb. 2010.

[2] A. Agarwal and M. Levy. The kill rule for multicore. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 750–753, San Diego, California, 2007.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI '10)*, pages 433–448, San Jose, CA, Apr. 2010.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 1–14, Big Sky, MT, Oct. 2009.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, 2009.

[6] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, 1996.

[7] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. DARPA IPTO, ExaScale Computing Study, `http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECS_reports.htm`.

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[9] T. Bingmann. STX B+ Tree, Sept. 2008. `http://idlebox.net/2007/stx-btree/`.

[10] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practices and Experience*, 18(9):807–820, 1988.

[11] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 43–57, San Diego, California, December 2008.

[12] R. Cattell. High performance data stores. `http://www.cattell.net/datastores/Datastores.pdf`, Apr. 2010.

[13] E. Chan, S. Venkataraman, N. Tkach, K. Larson, A. Gutierrez, and R. H. Campbell. Characterizing data structures for volatile forensics. In *Proceedings of the 2011 Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE '11, Oakland, CA, USA, 2011.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[15] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 74–83, Cambridge, MA, Oct. 1996.

[16] Christoph Rohland. Linux Kernel Documentation, tmpfs filesystem, Jan. 2001. `http://kernel.org/doc/Documentation/filesystems/tmpfs.txt`.

[17] J. Coburn, A. Caulfield, L. Grupp, A. Akel, and S. Swanson. NVTM: A transactional interface for next-generation non-volatile memories. Technical Report CS2009-0948, University of California, San Diego, Sept. 2009.

[18] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[19] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, Oct. 2009.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[21] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7:289–312, June 1994.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 205–220, Stevenson, WA, 2007.

[23] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[25] R. J. Feiertag and E. I. Organick. The multics input/output system. *ACM SIGOPS Operating Systems Review*, 6: 35–38, June 1972.

[26] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[27] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4):439–447, 2008.

[28] FusionIO, Sept. 2010. `http://www.fusionio.com/`.

[29] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, June 2005.

[30] Hewlett-Packard Development Company. HP Collaborates with Hynix to Bring the Memristor to Market in Next-generation Memory, Aug. 2010. `http://www.hp.com/hpinfo/newsroom/press/2010/100831c.html`.

[31] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, San Francisco, California, 1994.

[32] B. Holden. Latency Comparison Between HyperTransport and PCI-Express In Communications Systems. Whitepaper, Nov. 2006.

[33] International Technology Roadmap for Semiconductors, 2009. http://www.itrs.net/Links/2009ITRS/Home2009.htm.

[34] International Technology Roadmap for Semiconductors: Process integration, Devices, and Structures, 2007. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_PIDS.pdf.

[35] James Nelson. Linux Kernel Documentation, RAM disk block device, Oct. 2004. http://kernel.org/doc/Documentation/blockdev/ramdisk.txt.

[36] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pages 654–663, El Paso, TX, 1997.

[37] M. Kwiatkowski. memcache@facebook, Apr. 2010. QCon Beijing 2010 Enterprise Software Development Conference. http://www.qconbeijing.com/download/marc-facebook.pdf.

[38] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[39] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, pages 2–13, Austin, TX, 2009.

[40] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, pages 1303–1306, Washington, DC, USA, Apr. 2009.

[41] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 92–101, St. Malo, France, Oct. 1997.

[42] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and future directions for the scaling of dynamic random-access memory (DRAM). *IBM Journal of Research and Development*, 46(2-3):187–212, 2002.

[43] Mathieu Desnoyers. Linux Trace Toolkit Next Generation Manual, Sept. 2005. http://www.lttng.org/content/documentation.

[44] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, J. Luetzen, A. Orth, J. Nuetzel, T. Schloesser, A. Scholz, U. Schroeder, A. Sieck, A. Spitzer, M. Strasser, P.-F. Wang, S. Wege, and R. Weis. Challenges for the DRAM cell scaling to 40nm. In *IEEE International Electron Devices Meeting*, pages 339–342, May 2005.

[45] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, 2007.

[46] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, July 1999. ISBN 0521663504.

[47] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, pages 183–191, Monterey, CA, June 1999.

[48] Oracle Corporation. BTRFS, June 2009. http://btrfs.wiki.kernel.org.

[49] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43:92–105, January 2010.

[50] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 24–33, Austin, TX, June 2009.

[51] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Bangalore, India, Jan. 2010.

[52] S. Raoux, G. W. Burr., M. J. Breitwisch., C. T. Rettner., Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change random access memory: a scalable technology. *IBM Journal of Research and Development*, 52(4):465–479, 2008.

[53] Redis, Sept. 2010. `http://code.google.com/p/redis/`.

[54] O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3:2:1–2:27, February 2008.

[55] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2:277–288, November 1984.

[56] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[57] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottawa, Canada, Aug. 1995.

[58] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, Mar. 2003.

[59] Spansion, Inc. Using Spansion EcoRAM to Improve TCO and Power Consumption in Internet Data Centers, 2008. `http://www.spansion.com/jp/About/Documents/Spansion_EcoRAM_Architecture_J.pdf`.

[60] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24:412–418, July 1981.

[61] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (its time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 1150–1160, Vienna, Austria, Sept. 2007.

[62] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453 (7191):80–83, May 2008.

[63] Sun Microsystems. ZFS, Nov. 2005. `http://www.opensolaris.org/os/community/zfs/`.

[64] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004.

[65] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[66] VoltDB, Sept. 2010. `http://www.voltdb.com/`.

[67] R. C. Whaley and A. M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software – Practice and Experience*, 38(15):1621–1642, 2008.

[68] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. *The VLDB Journal*, 4: 629–673, October 1995.

[69] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 86–97, San Jose, CA, Oct. 1994.

[70] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 14–23, Austin, TX, June 2009.